TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MIKKO RANTALA

REAL-TIME COLLABORATIVE CODING - TECHNICAL AND GROUP WORK CHALLENGES

Master of Science thesis

Examiner: prof. Hannu Jaakkola
Examiner and topic approved by the Faculty Council of the Faculty of Business and Built Environment on 4th March 2015

ABSTRACT

MIKKO RANTALA: Real-time collaborative coding - technical and group work challenges
Tampere University of technology
Master of Science Thesis, 45 pages
May 2015
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiners: Professor Hannu Jaakkola and Dr. Jari Soini

Keywords: real-time, collaborative coding, synchronization, code integration, pair programming

This thesis explores the technical and group work challenges present in implementing and utilizing real-time collaborative coding. This method of programming allows multiple programmers to edit the same files concurrently, however the technical implementations must be carefully considered as they can have a negative effect on group work. Different methods of document synchronization and code integration are presented and evaluated in order to find an optimal solution for real-time collaborative coding. Two existing code integration methods and one theoretical method are examined for their effects on group work. The suitability and effects of real-time collaboration on programming are examined through pair programming, classroom programming and outsourcing.

The purpose of this thesis was to find optimal technical solutions among the various methods of implementing real-time collaborative coding and to evaluate the suitability of different models of programming in utilizing real-time collaborative coding. Certain superior technical solutions were found but some research issues still remain open.

Excerpts from a previously made research publication that focuses on use of CoRED, a real-time collaborative IDE made at Tampere University of Technology have been included. The excerpts provide additional information on logging and visualizing collaboration in a classroom environment.

TIIVISTELMÄ

MIKKO RANTALA: Samanaikainen ryhmäohjelmointi – tekniset ja ryhmätyön haasteet
Tampereen teknillinen yliopisto
Diplomityö, 45 sivua
Toukokuu 2015
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: Professori Hannu Jaakkola ja TkT Jari Soini

Avainsanat: samanaikaisuus, ryhmäohjelmointi, synkronisaatio, lähdekoodin integrointi, pariohjelmointi

Tässä diplomityössä tutkittiin samanaikaisen ryhmäohjelmoinnin toteuttamisen ja käytön aiheuttamia teknisiä ja ryhmätyön haasteita. Samanaikaisessa ryhmäohjelmoinnoissa useampi ohjelmoija voi muokata samaa tiedostoa samaan aikaan. Tällaisen ohjelmointitavan mahdollistavat tekniset valinnat täytyy suorittaa harkiten, koska niillä saattaa olla haittavaikutuksia ryhmätyöhön. Erilaisia synkronisaation ja lähdekoodin integroinnin malleja tutkittiin, jotta löydettäisiin paras mahdollinen vaihtoehto samanaikaiselle ryhmäohjelmoinnille. Kahta olemassa olevaa ja yhtä teoreettista lähdekoodin integroinnin mallia tarkasteltiin ryhmätyön vaikutusten kannalta. Samanaikaisen ryhmätyön vaikutuksia ohjelmointiin tutkittiin pariohjelmoinnin, luokkahuoneessa tapahtuvan ohjelmoinnin ja ohjelmointityön ulkoistamisen kannalta.

Tämän diplomityön tarkoitus oli löytää parhaita teknisiä ratkaisuja toteuttaa samanaikainen ryhmäohjelmointi ja arvioida eri ohjelmoinnin työtapojen soveltuvuutta samanaikaisen ryhmäohjelmoinnin käyttöön. Eräät tekniset ratkaisut havaittiin tarkastelussa parhaimmiksi mutta jotkin tutkimusaiheet jäivät ilman ratkaisua.

Tähän työhön on sisällytetty osia aikaisemmin suoritetusta tutkimuksesta, jossa tutkittiin Tampereen teknillisessä yliopistossa kehitettyä CoRED kehitysympäristöä. Sisällytetyt osat kertovat samanaikaisen ryhmäohjelmoinnin tarkastelun kirjaamisesta ja visualisoinnista.

PREFACE

This thesis was done as a part of the N4S project as a research assistant in the Tampere University of Technology.

I would like to thank Professor Hannu Jaakkola and Research manager Jari Soini for the opportunity to participate in this project, and for comments and feedback on the writing of this thesis. I would also like to thank all family and friends who encouraged me.

Pori, 18.5.2015

Mikko Rantala

CONTENTS

ABBREVIATIONS AND SYMBOLS

IDE	Integrated Development Environment
OT	Operational Transformation
RAM	Random-access memory

.

1. INTRODUCTION

Real-time collaborative coding is a new technology in software engineering that has emerged within the last fifteen years due to improvements in the quality and pervasiveness of Internet connections and the development of web technologies that have enabled reliable real-time communication via browsers regardless of geographical distance. In real-time collaborative coding two or more programmers at each their own workstation can all work on the same file at the same time. This differs from traditional software engineering where coding is mostly solo work at a single workstation. This new way of collaborating is not without its problems as coding is often seen as a solitary activity with clearly defined territories between programmers, some programming concepts such as code ownership are entirely absent from real-time collaborative coding. A shift in perspective is required to become more accustomed to collaborative programming; good communication skills become even more important. Certain real-time collaborative programming environments also place constraints on programmers due to certain technical choices; while the editing part of coding could be considered a more or less trivial problem due to the multitude of solutions, compiling a program is a non-trivial concern in implementing real-time collaboration in an integrated development environment (IDE).

The research problem of this thesis is the implementation and utilization of real-time collaborative programming in and IDE and in different programming environments.

- *The first research question* is derived from the implementation part of the research problem: what kind of technical problems and solutions exist to enable real-time collaborative coding and running a program? Compiling and running a program can be considered a trivial problem for traditional IDEs, however adding the capability for real-time collaborative coding complicates the process of compiling a program. This research question was further divided into two main areas: document synchronization and code integration. These two topics are considered separately and are evaluated in their own categories. In both topics a "best solution" is sought based on difficulty of implementation and technical reliability.

- *The second research question* comes from the utilization of real-time collaborative programming in different programming models of working; pair programming, classrooms and outsourcing are examined as targets of real-time collaboration. The effects of the technical solutions presented in the first research question are also evaluated for their impact on users; a "best solution" is sought on the basis of least negative impact on user experience.

Source material for this thesis has been mainly gathered from publications and articles on the topics of collaboration, real-time collaborative IDEs, document synchronization and studies on the effects collaboration has on programming and programmers. Constructive research was chosen as the research method because the purpose of this thesis is to provide an overview of real-time collaboration in programming, and aggregate some technical and group work challenges and solutions to date that emerge in implementing real-time collaborative programming. Although this type of programming method is unlikely to be a silver bullet, a study on the various technical implementations and ways of working in order to find optimal solutions and potential pitfalls can yield valuable insights for the future.

Real-time collaborative editing and web-IDEs capable of real-time collaborative coding are explored in Section 2 along with IDE examples and screenshots. Traditional and real-time collaborative coding differ from each other in many ways, technical challenges such as document synchronization between users, code integration and compiling have been solved in different ways in different IDEs with varying success (Fraser 2009) (Goldman et al. 2011). These technical challenges and various solutions are presented and examined in Section 4. Collaborative coding in real-time offers new possibilities in ways of working in the fields of education, and amateur and professional software development. Real-time collaboration has already been possible via remote viewing of another person's desktop; now coding problems can be solved or coached through in a more direct manner via real-time collaborative coding implemented in IDEs. The effects of collaboration in both traditional pair programming, and real-time group work have been studied before (Nosek 1998; Williams et al. 2000; Nawrocki et al. 2001; Kilamo et al. 2014). These issues and the results of these studies are examined in Section 5. Section 6 contains excerpts from a study made of a real-time collaborative web-IDE's log data. The IDE in question, CoRED (Lautamäki et al. 2012) has been developed at Tampere University of Technology, and it has been tested during two different code camp events. This section provides a look at logging and visualizing collaborative coding. Additionally issues such as knowledge transfer during collaboration and utilizing visualizations to detect bad coding habits are considered. Sections 7 and 8 contain discussion and a summary, and end the thesis.

2. REAL-TIME COLLABORATION

Collaboration in programming means working together. There is an important difference between collaboration and the traditional style of software development which is cooperative. Dillenbourg (1999) defined the difference between cooperation and collaboration as follows: “In cooperation, partners split the work, solve sub-tasks individually and then assemble the partial results into the final output. In collaboration, partners do the work 'together'.” Collaboration is thus a social activity that both requires and teaches communication and teamwork skills. Collaboration is also not limited to sending and receiving instructions, acknowledgements and questions between team members of different hierarchies but includes negotiation and sharing of knowledge (Roschelle et al. 1995) between collaborators who have a shared responsibility of their work. The sharing of knowledge is a crucial point in collaborative problem solving; different team members possess information that other team members might not have and are capable of combining them to find an optimal solution (or a new one) through negotiation.

Real-time collaborative editing first emerged to a wider audience during Douglas Engelbart’s 1968 demonstration now known as “mother of all demos” (Engelbart et al. 1968). It wasn’t until the beginning of the 90s that editors started to become widely available. The development of web 2.0 technologies, specifically Ajax, allowed real-time collaborative editors to be implemented in webpages. The ease of access and use compared to installed editors enabled a growth in popularity of which the most known example today is Google Drive.

A large number of web IDEs with real-time collaboration capabilities exist. Their features that relate to collaboration are mostly the same with some small variations: all of them possess the capability to allow multiple users to edit a file at the same time and users are identified by a nametag or a colour attached to their cursors. After a programming experiment held at Tampere University of Technology utilizing CoRED (Nieminen et al. 2013) a list was compiled of features that should be included in a real-time collaborative IDE. Table 1 below contains a list of the suggested features.

Table 1 *Important features for a collaborative web based IDE (Nieminen et al. 2013)*

Feature	**Implemented in CoRED**
Edits should be visible in real-time	Yes
Possibility to deploy a project containing errors	No
Project management tools	No
Single click deployment	Yes
Sufficient awareness between users	Partly
Support for undo/redo	Partly
Tools for communicating	Partly
Tools for debugging and testing	No
Version control support	No

The first feature on the list is the bare minimum requirement of a real-time collaborative editor and as such should be a self-evident inclusion. The second feature, possibility to deploy a project containing errors, relates to the technical problem of code integration and is further explored in Section 4. Features relating to awareness between users and tools for communicating are explored in Section 5.

Real-time collaborative coding as a feature requires additional technical work to implement. Edits made by multiple users to the same source code file are a problem for IDE features that require an error free source code to function, these features include compiling, debugging and running the program. The purpose of this thesis is to examine the unique problems that arise when real-time collaborative coding is implemented and utilized in a web IDE, specifically two research questions have been chosen. The first question is: what kind of technical problems and solutions exist to enable real-time collaborative coding and running a program? The second question is: how does implementing collaborative coding in real-time affect group work?

The first question is further defined to account document synchronization, merge conflicts, source code integration and running the program. These problems are examined in Section 4. The second question examines what effect technical implementations of real-time collaboration have on group work and how real-time collaboration changes group work compared to traditional software development. Effects on group work are examined in Section 5.

2.1 Real-time collaborative IDEs and editors

A large number of IDEs and editors with real-time collaboration capabilities are available; most of these are browser-based although Eclipse features a plugin called Saros (http://www.saros-project.org/) that enables real-time collaborative coding. Table 2 below lists a number of IDEs and editors that feature real-time collaborative coding and editing. The entries below are IDEs unless otherwise mentioned.

Table 2 *Real-time collaborative IDEs and editors*

Name	Source	Additional information
Cloud9	https://c9.io/	Monthly subscriptions increase storage and computing capacity
Codassium	http://codassium.com/	Interview focus with built-in video conferencing
Codeanywhere	https://codeanywhere.com/	Also available on iOS and Android
Codebox	https://www.codebox.io/	Free account limited to one collaborator
Codenvy	https://codenvy.com/	Free account has limited build and runtime
Coderpad	https://coderpad.io/	Interview focus with session playback
Codiad	http://codiad.com/	Installed to own server, donation option
Codio	https://codio.com/	Education oriented with curriculum integration
Collabode	http://groups.csail.mit.edu/uid/collabode/	Eclipse server provides IDE services
Koding	https://koding.com/	Collaboration feature is “still under development”
Kobra	https://kobra.io/#/	No IDE features
Nitrous	https://pro.nitrous.io/	Free account limited to two hours per day
ShiftEdit	https://shiftedit.net/	No free account
Squad	https://squadedit.com/	Free account utilizes only local files

There is no shortage of real-time collaborative IDEs and editors to choose from. The table above contains mostly IDEs because editors are mostly a trivial implementation requiring only document synchronization. Most of the IDEs listed above feature free, restricted accounts. Monthly and yearly subscriptions increase the available cloud storage and memory capacities. The popularity of web implementations is most probably due to the zero-setup nature of such IDEs. Where desktop IDEs require either at least a plugin or a local client-server network installation, web-IDEs provide much the same functionalities with a login and a subscription. This lends web-IDEs a highly mobile nature that does not require a programmer to relocate or use a virtual private network program to connect to a company’s network.

2.2 Example implementations

Gathered below are three web-IDEs and one web-editor that have been chosen as a sample to demonstrate collaboration features, specifically sharing access to a project, awareness of other users’ actions, and communication tools. Implementations of access sharing in web-IDEs range from a simple sharing of workspace via web address to an invite based

system with individual read/write rights. Awareness of other users' actions is most commonly highlighted with colored cursors matching to specific users that may contain a nametag. Most IDEs and editors also include a text chat system for the bare minimum of communication although some have video transmission built-in. Access sharing and read/write rights are an important issue for the outsourcing model of programming which is discussed further in Section 5.

Cloud9

Released in 2010 Cloud9 is a web-IDE with a more professional façade that offers a restricted free mode with a range of paid options that increase the amount of available workspaces, and server random-access memory (RAM) and disk space. The editor for Cloud9 is Ace (http://ace.c9.io/), which is an embeddable code editor that originally began as the Mozilla Skywriter project. Cloud9 offers a fairly robust system for inviting collaborators, and different levels of access and viewing rights. Figures 1 and 2 below from the Clou9 website illustrate both the invitation system and the editor.

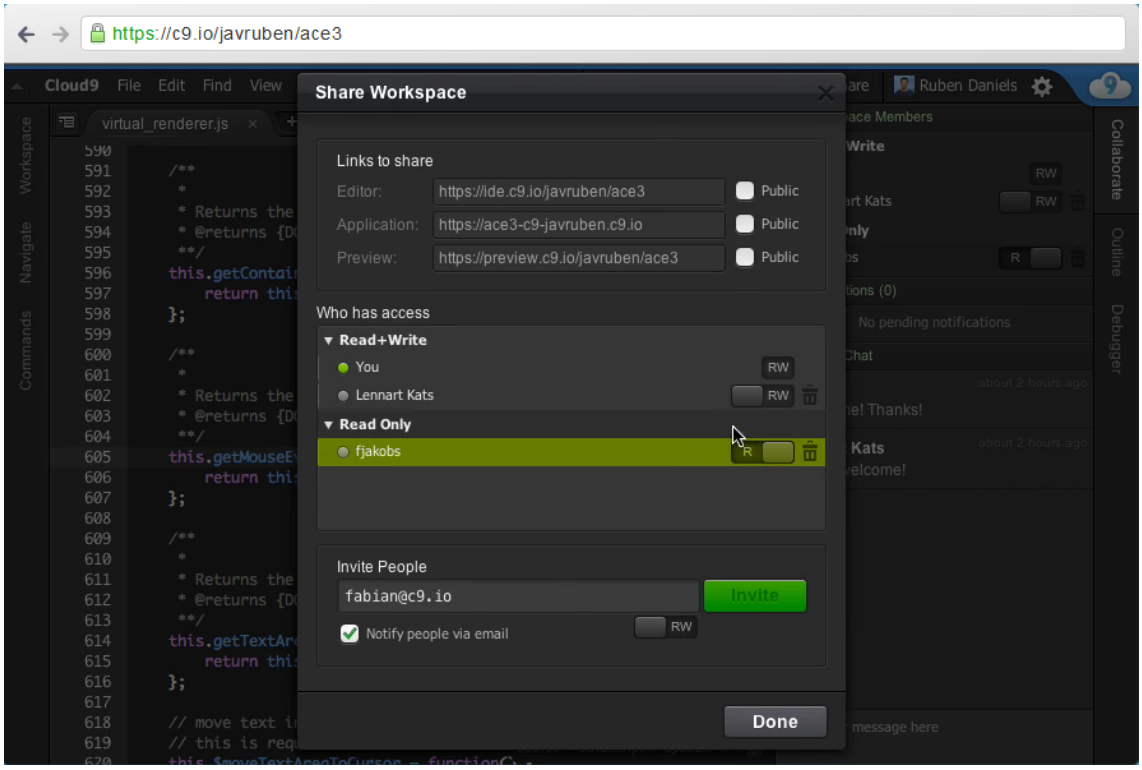

Figure 1 *Cloud9 workspace sharing options.*

The figure above shows the workspace sharing settings available in the Cloud9 web-IDE. The first section allows for sharing access via a web address to the project editor, application or a preview of the application. Second section is used to set read/write rights for collaborators that have been invited from the third section.

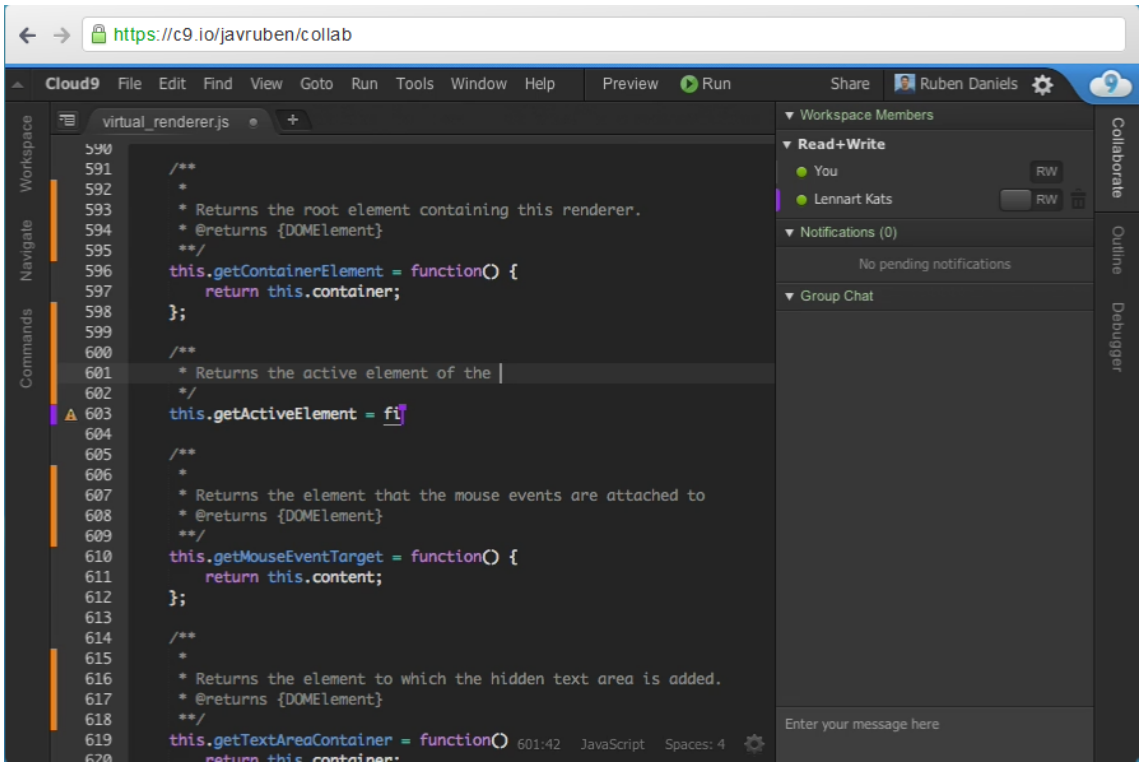

Figure 2 *Cloud9 collaborative editing.*

In the figure above two users are collaborating on a single file. User Daniels has invited user Kats to his workspace as seen in the upper right corner. In the editing window cursors are active in rows 601 and 603 with the guest editor's cursor colored purple. Persistent colored markers allow collaborators to retain awareness of other users' current focus.

Kobra

Kobra is strictly an online editor and does not contain IDE features such as compiling. This web-editor features a more primitive collaboration and access system than Cloud9. Editor access is granted with a web address and anyone who is given a link to the file can join to edit it. This type of collaboration system lacks granularity in user rights. An advantage to this system is speed and ease of use, inviting other users requires only a few clicks. Figure 3 below illustrates the editor view with guest cursor highlighting.

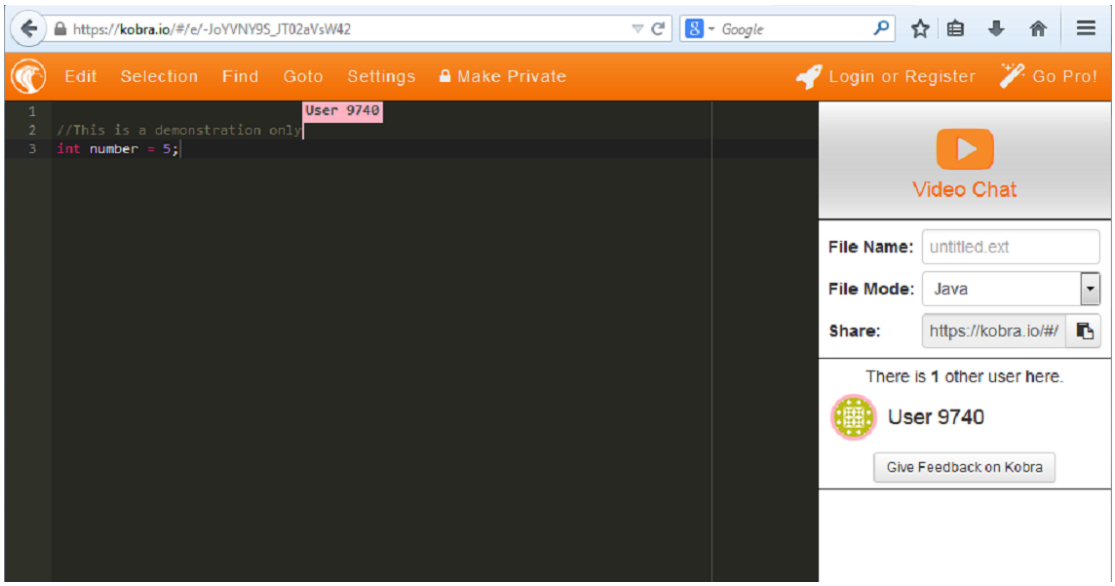

Figure 3 *Kobra collaborative editing.*

A guest user by the name of User 9740 has been invited via a shared web address. A guest's cursor location is highlighted with a colored nametag while editing and a few seconds afterwards.

Coderpad

A real-time collaborative web-IDE marketed as an interview tool for engineering candidates with a playback feature that records and plays the entire editing and execution of a session. Figure 4 below shows a typical editing view with two users collaborating.

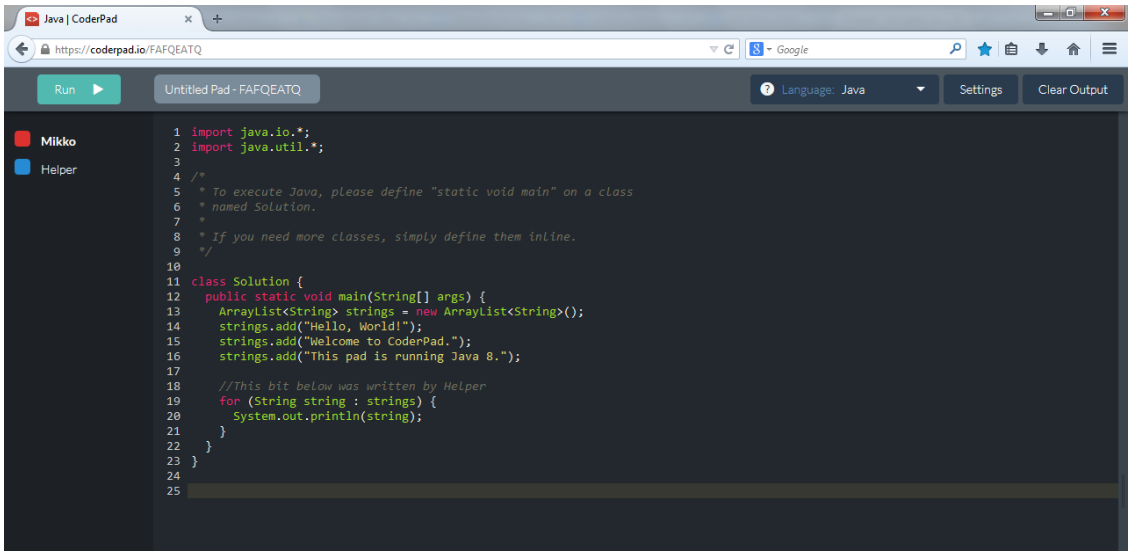

Figure 4 *Coderpad collaborative editing.*

Coderpad's editor does not feature constant highlighting of user cursors, active editing shows a colored cursor matching the user in the left sidebar.

Codassium

A collaborative IDE with built-in video conferencing also marketed as an interview tool. Figure 5 below shows an editing session with video conferencing.

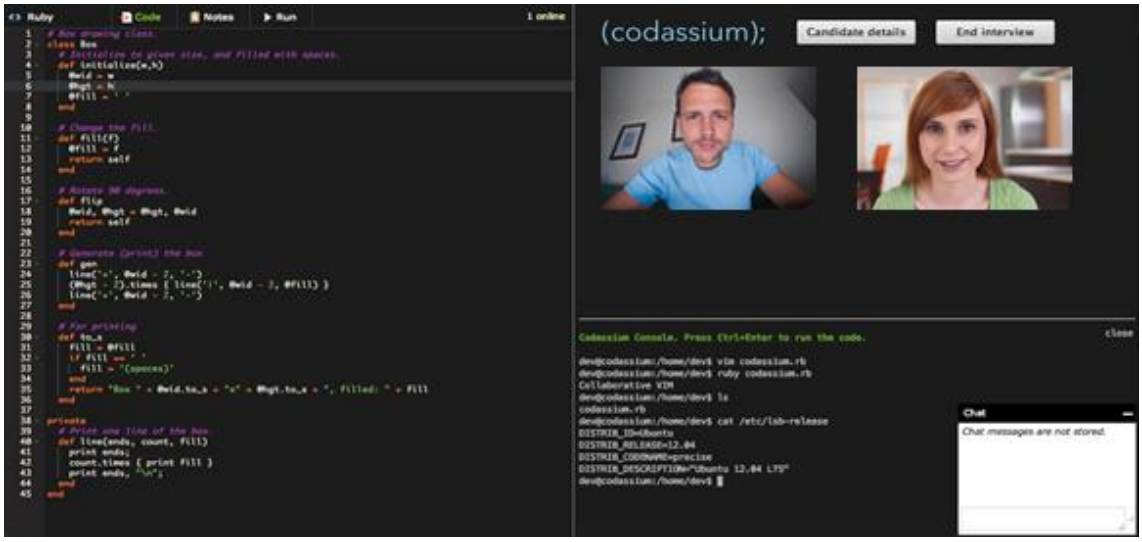

Figure 5 *Collaborative editing with video conferencing.*

This implementation is configured for an interview setup between two persons. This pair approach is a simpler to implement than real-time collaborative coding for three or more persons.

3. RESEARCH METHODOLOGY AND MATERIALS

As this thesis is meant to provide an overview of technical and group work issues discovered so far and determine optimal solutions to them based on existing ones, constructive research was chosen as the most suitable method.

The research process began with a study of the surrounding theory and existing practical issues and solutions. Theory was derived from studies on both pair programming and real-time collaborative coding. The majority of available publications focus on pair programming which is not surprising considering real-time collaborative coding is not as well-known. Information regarding technical solutions was found in publications on real-time collaborative IDEs specifically CoRED (Lautamäki et al. 2012) and Collabode (Goldman et al. 2011).

Studies and experiments on different models of programming such as pair programming and outsourcing were also examined. There are not many publications regarding the issue of fitting real-time collaborative programming into different models of programming, thus this topic provided the most "freedom of thought". Additionally an earlier research publication on CoRED included in parts to highlight visualization of collaboration in a classroom environment.

4. TECHNICAL CHALLENGES AND SOLUTIONS

Implementing real-time collaborative coding in an IDE is not a trivial task, and requires solving at least two problems: synchronizing document changes across all users and compiling the program while others might still be editing source code. Synchronization algorithms have to deal with issues such as unreliable network connections, guaranteeing constant user access, and ensuring the right order of applying edits form other users. Code integration and compiling is complicated by the presence of collaborators which necessitates handling syntax errors from edits made by other users. These technical challenges have multiple solutions that often make tradeoffs between ease of implementation and usability; this is especially true for compiling. This section details problems and solutions to the above issues, some of them may not be in use or theoretical but have been included to round out comprehension.

4.1 Document synchronization

One challenge of a successful real-time collaborative IDE is synchronizing two or more documents in real-time in order to maintain constant awareness of the state of the documents being edited. This section details some problems and solutions of synchronizing documents across a network of multiple users. These solutions are further evaluated in Section 5 (Group work) and 7 (Discussion). Sun et al. detail three inconsistency problems that must be solved in designing and implementing a real-time cooperative editing system: divergence, causality violation, and intention violation (Sun et al. 1998).

A system that allows for real-time concurrent editing at multiple locations will lead to **divergence** meaning that local copies of the same document will have differences since distributing edits to other users takes time due to latency. Delays in distributing edits due to latency also create a new problem in the form of commutativeness of edits. Often an operation, that is executed locally and then sent to other clients, will have the wrong order due to edits made at the receiving end changing the layout of the document. This problem is illustrated in figure 6 below in the section on event passing.

Causality violations can occur due to latency changing the order of arriving operations. Should a local operation be a response to a previous remote operation that arrived from another user these operations, if transmitted in a different order to a third user, could cause confusion due their wrong order of arrival.

Whereas causality violation deals with the order of arriving operations, **intention violation** deals with the actual effect of operations at the time of their execution. Figure 6 below illustrates a possible intention violation where two concurrent operations, if executed as is, would result in divergent documents.

Common synchronization methods are locking, event passing and three-way merges, they are presented below based on Neil Fraser's summarization of them (Fraser 2009). In **locking** only one user may have write access while others can only read the file. This method could be enhanced by locking only portions of a document however for small documents this would be a hindrance in editability. Another problem would be the transferring of write access, complex safeguards or watchdog timers would have to be built to ensure that write access would not be lost due to a network communication error.

Event passing is a technique that captures user actions and mirrors them across the network to other users. A popular event passing implementation is Operational Transformation (OT) (Ellis et al. 1989) that allows for instant local edits that are distributed to all other users. OT works by transforming concurrent operations so that when they are sent to other clients they perform correct changes in the new context and result in the same document on all clients. Figure 6 below illustrates a simple edit case.

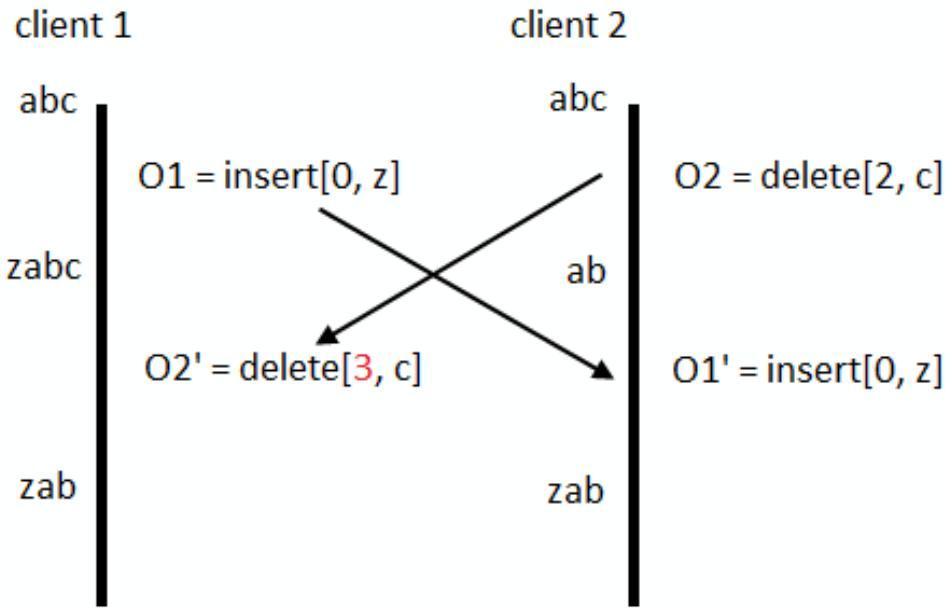

Figure 6 *Transforming an operation to ensure correct execution.*

The initial document contains a string of text "abc" for both client 1 and 2. Both clients edit their respective documents at the same time. Client 1 inserts a "z" character to the beginning of the string resulting in "zabc" while client 2 removes the "c" character resulting in "ab". Both of these edits must be sent to other clients in order to maintain synchronization however a transformation must be performed since Operation 2 (O2) as is would result in "zac" because client 1's edit has already changed the original string. With the transformation from O2 to O2' the index for the delete operation is incremented to account for the change in index value caused by Operation 1 (O1) and both documents end up containing the correct string "zab". Operation 1 is also transformed but in this example there is no functional difference between O1 and O1'.

While obtaining a snapshot of the state of a document is easy, modern user interfaces allow for a multitude of user actions such as cut, paste, drag and drop that further complicate synchronization. A failure to apply an edit made by a different user will result in a forked document, which can have catastrophic effects since any further edits may be incorrectly placed resulting in an even greater difference in document synchronization.

Three-way merging utilizes a server that receives document contents from each user and performs a difference analysis between each user's document and the base document stored on the server. After the merge is complete a new base document is sent to all users. This process is illustrated in figure 7.

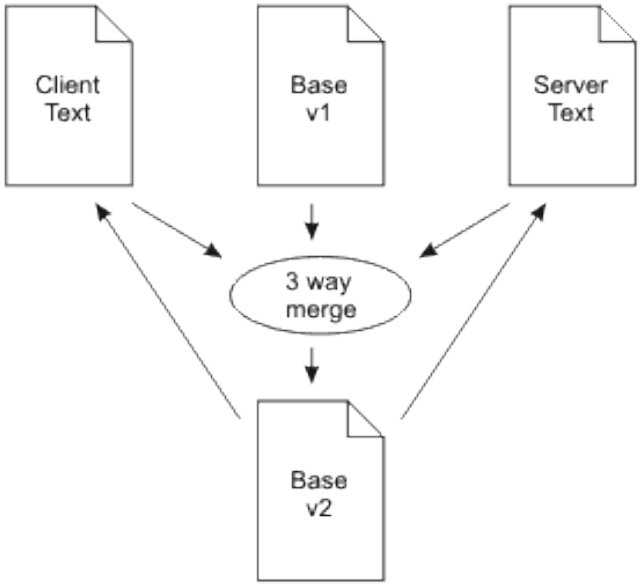

Figure 7 *Three-way merge (Fraser 2009).*

A major problem with the three-way merge is the half-duplex nature of the system, as long as a user keeps typing no changes are received, however once the user stops typing input from others users is either integrated successfully or a collision warning dialog appears. Since real-time collaborative coding relies on users being aware of any concurrent editing being made to avoid editing the same lines, this type of latency in integrating changes will most likely prove undesirable.

Differential synchronization is a method developed by Neil Fraser for document synchronization (Fraser 2009). Unlike three-way merge differential synchronization has been designed to operate constantly allowing users to keep editing and still receive changes from other users. Figure 8 below illustrates the synchronization cycle of the differential synchronization algorithm.

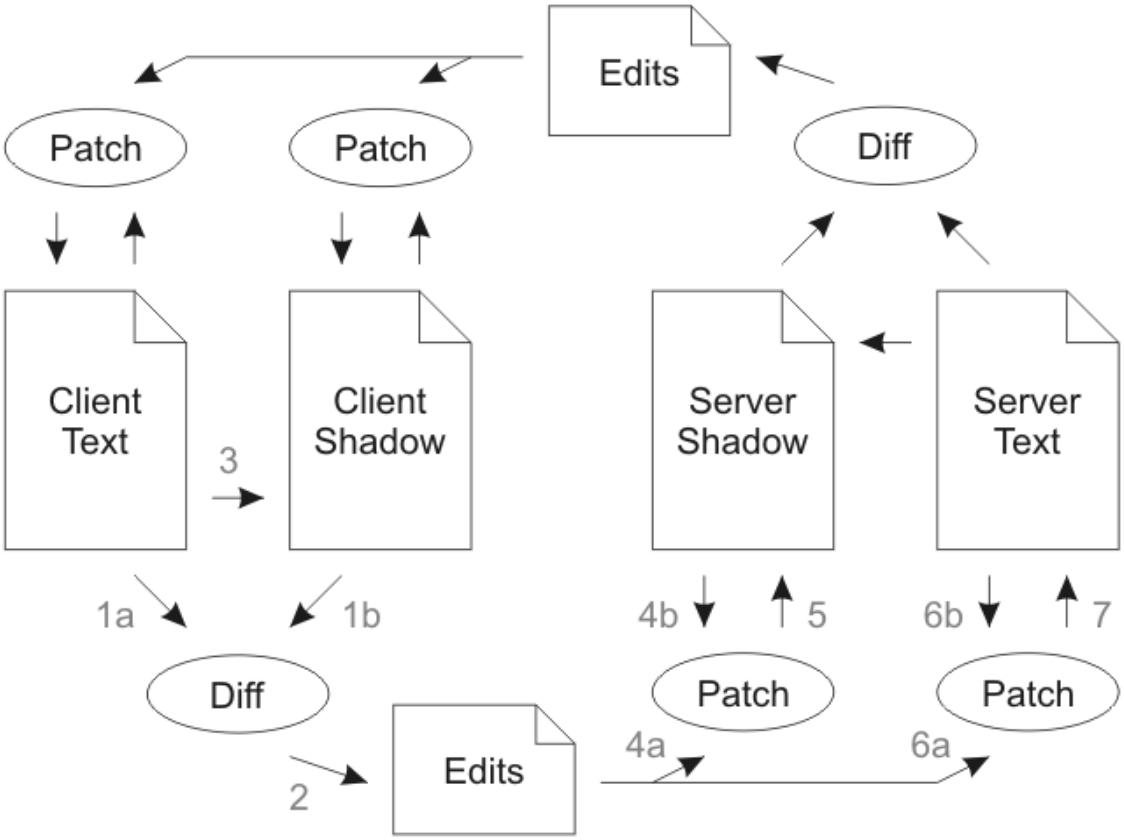

Figure 8 *Differential synchronization with shadows (Fraser 2009).*

Let us assume that the starting situation in a network is idle with all text and shadow files being identical. Both client and server text can be edited at any time. The below description is for a client-side edit to server-side patch but the process works in the same way for server to client updates.

1. Client Text and Shadow are compared for any differences.
2. A list of edits performed on Client Text is recorded.
3. Client Text is copied over to Client Shadow. This copy is identical to Client Text in step 1.
4. Edits are applied to Server Shadow.
5. Server Shadow is updated.
6. Edits are applied to Server Text.
7. Server Text is updated.

After this process is complete Client Text and Server Shadow must be identical to each other (or Server Text and Client Shadow if the edit originated from the Server). A checksum of Client Shadow is sent with the Edits and compared to Server Shadow after the update process is complete. This is done to verify that the update process completed successfully. If the checksum fails to verify, the entire text must be transmitted to synchronize. In practice differential synchronization does not transmit edit operations, instead it gathers diffs between files and distributes those.

4.2 Source code integration and compiling

A unique problem for real-time collaborative IDEs is how to handle integrating (merging) changes made by multiple users as they happen from a source code compilation process viewpoint. Merging source code means updating a file with changes from multiple users. These changes may sometimes conflict with each other if for example edits have been made to the same line. As an example let us assume that Alice and Bob are both using a non-collaborative IDE to edit the same file. Both of them download a copy of the file to be edited from their repository. Alice proceeds to implement features to a function that was previously only a stub while Bob also edits the same lines as Alice. Once both Alice and Bob are satisfied with their work they upload their file back to the repository. A conflict is detected since both Alice and Bob edited the same lines, at this point the conflict is usually resolved by a person selecting either Alice's or Bob's version as the correct one. This conflict occurs in non-real-time collaborative IDEs because users might not be aware of each other's work on the same file.

In a real-time collaborative IDE however the above situation should not occur because edits made to the same file are shown to all who are editing the file in real-time and thus awareness of any edits made by others is immediate. The real problem becomes handling compilation since due to divergence (presented in Section 4.2) local documents are in a shifting state where syntax errors may exist and compiling is blocked. There are a number of ways to approach this problem; three different solutions are explored below. These solutions utilize the concept of different types of source code copies as described by Goldman (2012).

- Master copy: source code used to compile and run the program.
- View copy: source code that appears in the user's editor.
- Working copy: source that the IDE uses to generate syntax error warnings and code completion.

Automatic code integration merges any edits into a single copy is a combination of master, view and working copy, whether they contain syntax errors or not. In this method what the user sees in the editor is what will be compiled if possible. In practice this means that in order to compile the program the source code must be brought to a state where no errors exist and if more than one user is editing everyone else's work is disrupted, since they will be have to either complete their editing or comment out parts to remove syntax errors. As an example let us assume that Alice and Bob are both editing the same file concurrently. Alice finishes her work first and wants to test her changes but she cannot do so until Bob brings his work to an error free state so Alice can compile the program. This method is used by the CoRED web-IDE (Lautamäki et al. 2012). Technically this is the least complicated method to design and implement but the most cumbersome for users to work with.

Manual code integration requires a user to manually decide when to commit error free code to a master copy from which the program is compiled. This would allow users to maintain a syntax error free copy thus enabling compilation of the project at will without disrupting anyone else's work. The immediate downside to this kind of method is the lack of knowledge of exactly what features does the master copy contain. Committing code could also become a disruptive activity if other users are actively editing the file(s) being committed. In this method the view and working copies are the same.

Error-mediated integration is a method developed for the Collabode (Goldman et al. 2011) web-IDE that automatically integrates error-free source code to a master copy. In this method there are three copies of the source code; a master copy, the editor view copy which is the same for all users and the working copy that the IDE uses to report syntax errors for the local user. Code being edited simultaneously will still be visible in the editor to all users however the code being edited will not be subject to error checking or code completion for other users until it reaches an error-free state (Goldman et al. 2012). This method also contains the problem of users not being fully aware of exactly what will be compiled if others are still editing incomplete code.

5. GROUP WORK AND REAL-TIME COLLABORATIVE CODING

Programming has historically been solo work at a single work station. Even though the size and complexity of software has increased, tasks are divided between programmers who each perform their work on their own. The advent of agile software engineering gave rise to a greater emphasis on internal communication and collaboration where team members were kept up to date on the overall progress of the project. One agile methodology, Extreme Programming (Beck 2004), brought pair programming to the fore as a core practice, a method which has two programmers collaborating at a single workstation. Modern technology however has enabled real-time collaborative coding, a method by which multiple programmers can simultaneously edit the same file. Pertinent questions are where can real-time collaborative coding be utilized, and what gains can be derived from it? This section explores these questions from multiple angles; education, and amateur and professional programming are considered. Additionally the effects of the technical solutions, listed in Section 4.2, on group work are considered further. The concept of knowledge space is also used in this section, it refers to the sum of information a person has on a particular topic, and which they can share with others whose knowledge space on the same topic might overlap to different degrees. Figure 9 below illustrates this concept.

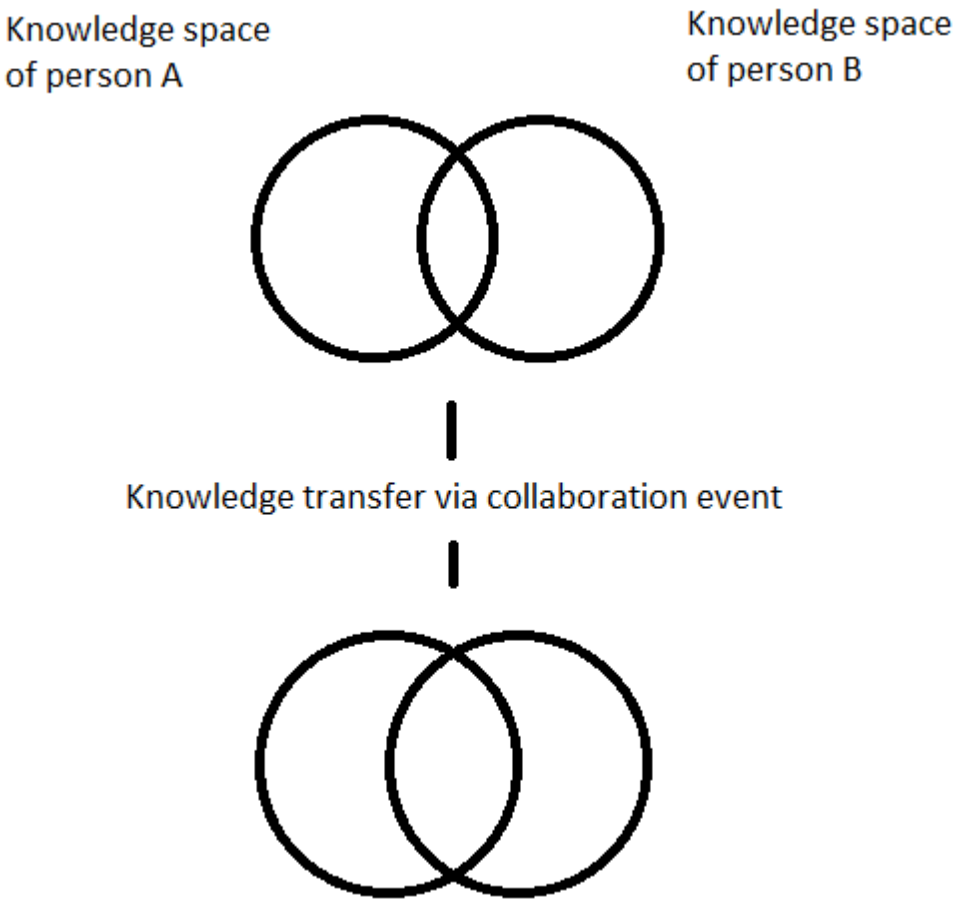

Figure 9 *Knowledge space transformation during collaboration.*

Let us assume that two persons, Alice and Bob, are collaboratively solving a programming problem. Each of them has a set of programming related information which we will name "knowledge space". These knowledge spaces are most likely different between Alice and Bob due to education and experience but there is a certain amount of overlap as illustrated in the figure above. In order to solve a programming problem both Alice and Bob utilize a subset of their knowledge space which contains information relevant to the task at hand. Alice and Bob can solve the problem in a number of different ways, and if the problem is not a trivial one, in an ideal collaboration scenario they would first communicate different solutions to each other and evaluate their choices for the best solution. Their collaborative problem solving activities helps expand their knowledge space. The usefulness of collaborative problem solving, in terms of knowledge transfer, hinges on the difference in the abilities of the collaborators. Experts are less likely to benefit from this kind of arrangement amongst each other as they are more likely to be capable of producing a solution on their own. The relationship between the usefulness of collaboration and the abilities of the collaborators is further explored in the sections below.

5.1 Ways of working

Pair programming

Pair programming is a practice in which one person writes code and the other acts as an observer who constantly reviews the code being written. This happens at a single workstation with the programmers switching roles periodically. This kind of programming method has several advantages; the amount of knowledge available to solve the problem is increased, even if there is likely to be overlap in the two programmers' skills. The team members' communication skills are likely to develop as a result of hashing out problems together. Pair programming as an experience hinges quite a lot on the skill levels of the participants (Lui et al. 2006); let us consider two distinct levels: rookie and veteran. Ideally the rookie should be the one writing code with the veteran mentoring them through the process. A danger lies in the veteran taking over the reins due to frustration or lack of engagement on part of the rookie. Pairing two rookies in a professional environment is likely unwise for critical tasks which would probably not be handed to a rookie programmer in the first place. A more likely use for a pair of rookies is in an educational setting; it is likely that participants are all less experienced than those already established in industry. Pairing veterans might prove less useful as well unless the problem at hand requires cross-domain experience since otherwise the veterans might both be capable of solving the problem on their own (Lui et al. 2006).

Since pairs do not produce code at double the speed of solo programmers man-hours will be higher, however this should be balanced against the generally higher quality and faster production speed of source code. Pair programming is not a general purpose answer to programming work division; it can however be an accurately deployed solution when higher quality and speed matter.

Multiple studies of the effects of pair programming have been made; Nosek (1998) investigated the performance of pairs versus individuals and found that pairs produced more readable and functional solutions. Lui et al. (2008) reported that pair programming performed the better the harder the task was. Williams et al. (2000) found that pairs were generally faster and had greater confidence in their work. Cockburn et al. (2000) reported shorter code length, less defects, and better overall understanding of the project among team members. A lower defect count can also lead to lower maintenance costs as less bug fix requests are filed. Expanded awareness and direct exposure to different parts of the project among team members lessens project risks from being affected by the bus factor meaning what would happen to the project if person A were to be unable to continue with the project?

So how does real-time collaboration fit into pair programming? A natural fit is remote pair programming in which the pair is in separate locations, this method does not necessarily follow a typical coder and observer/advisor structure but might have both members

actively code. More preparation is required to support this kind of coding method; audio and video communication software should be utilized to maintain optimum communication capabilities along with a shared memo tool to keep track of progress and issues. Nowadays there is no shortage of video conferencing software and real-time collaborative coding capable IDEs thus engaging in this form of collaboration is only a matter of setup and matching timetables. Flor (1998) suggests that in order for remote pair programming to have the same benefits as close-proximity pair programming, the remote pairs must exhibit the following properties: search through a larger space of alternatives; efficient communication; ongoing sharing of goals and plans; joint production of ambiguous plan segments; reuse of system knowledge; shared memory for old plans; and the ability to dynamically incorporate new divisions of labor and collaborative interaction systems. Flor (2006) also argues that remote collaboration must be able to replicate the same environment as close-proximity pair programming, where participants can influence the work with not only obvious communication such as talking and observing another's screen but more subtle ones such as gestures not necessarily meant to communicate anything but which might cause a participatory reaction.

Classroom environment

Education is also being reshaped by modern technology and the shift towards the web; massive open online courses (MOOC) utilize the web as a social and collaborative platform. A traditional classroom environment in teaching programming however focuses on transferring knowledge from the teacher to the students. Web courses that feature videos, slides or other material can provide students with resources that are important to learning however they do not instill motivation (Stahl 2006). In such an environment collaborative learning activities are left to the students' own choice or are simply a problem solving support activity. Classes that feature practical coding typically have the instructor either sitting at their desk waiting for students to call for help or walking around the classroom checking on students and asking if they need help. In a close-proximity environment this type of teaching activity is sufficient but courses existing in a web environment require additional support to retain the same information transfer infrastructure as a classroom environment. A pertinent question then is can this infrastructure be, if not enhanced then at least retained by technology? Screen sharing tools allow teachers a remote view into a student's problem solving process thus replicating a part of a close-proximity classroom environment, however real-time collaborative IDEs can take this a step further by allowing interaction via free document navigation that is not restricted to the student's view, and editing to present correct or alternative solutions. Perhaps more importantly this also allows students to observe and learn from their peers, which increases the available knowledge space to encompass the entire group.

Like pair programming collaborative coding in a group highlights communication and working as a group. Imparting skills in those areas, in addition to programming ability, to students is important in preparation for moving to an industry environment where the

ability to lead, take orders, and work as part of a team can make or break chances at employment.

Outsourcing

Programming projects can benefit from real-time collaborative capabilities in the form of outsourcing parts of their work. Web-IDEs capable of real-time collaboration usually allow invitation of collaborators either via sharing a web address or by direct account based invite as demonstrated in Section 2. More sophisticated IDEs also allow limitations to be put on the collaborator's rights such as specifying an area of source code that cannot be edited or a file specific access restriction. These controls provide a powerful tool for allowing an outside programmer access to a very specific problem area, combined with real-time collaboration this allows the project members to retain awareness on the state of the work and more easily provide information on the problem area. Bringing an outside person into the project or at least to a very specific part of it also helps keep overall control of the project within the original team instead of outsourcing an entire module or program where retaining awareness of the state of the project can be problematic. Another advantage of this approach is being able to immediately utilize any in-house testing procedures on changes made by outsourced programmers; since the source code is not housed in a separate repository there is no need for elaborate arrangements to test changes. While this type of micro-outsourcing might not overall be as cheap as outsourcing the development of a project wholesale (which is by no means guaranteed) this kind of approach allows for faster testing and integration, better awareness of the status of the project, and greater transparency on the work being done.

The greater freedom of maneuverability in hiring programmers that globalization has created is further enhanced by real-time collaborative capabilities however this model of work may not be as suitable or easy to adopt for all cultures as wholesale outsourcing. The rules and tacit knowledge that govern social interaction can be radically different between some cultures which can lead to misunderstandings or unnecessary friction in human interaction. In some cultures a verbal request to do something can be interpreted as a direct command or simply a suggestion. Geographical distances can also mean differences in working times which can prove to be a hindrance in a collaborative scenario whereas wholesale outsourcing is not as affected. An additional consequence of geographical distance can be legal differences, tax and worktime issues must be considered beforehand to avoid potential legal trouble. Outsourcing remains both a risk and a cost reducing opportunity as long as careful preparation is taken (Boehm 2006).

5.2 Effects of technical choices on group work

Certain technical solutions to implementing real-time collaborative coding in an IDE can have an impact on group work. The solutions explored in Section 4 were divided into synchronization and code integration methods, of these two synchronization is the lesser

problem for the following reasons; a common web-IDE synchronization solution is Operational Transformation which behaves well unless network latency is high (which increases the chances of packet loss, and need for retransmit for all synchronization methods); synchronization is a background activity that does not require active user attention thus allowing focus to be applied to solving actual programming problems. Assuming that document divergence is not a problem, the most disruptive synchronization related factor becomes the smoothness of applying incoming edits. A graceful way of integrating edits by others would be to mimic typing instead of pasting chunks of text.

A substantial problem however is presented by the method of integrating code to be compiled. This problem refers to the fact that in order to test the program it has to be compiled; in order to compile a program it is best to compile the source code without errors which will almost certainly be present in the project with multiple active collaborators. Presented in Section 4 were three different code integration methods; automatic, manual and error-mediated. These methods have a disruptive effect on either the whole group or a single member due to different reasons.

The first to be considered is automatic code integration in which there is no separation between editable and compiled code. A user that wants to compile and test the program must first bring the project to an error free state; this requires all other users to either complete their current task or edit their section so that it is incomplete but contains no errors. The first option requires waiting an unspecified amount of time until everyone is ready, not only would the person wishing to the test the program have to wait, so would everyone else who would be forced to wait for the last one to complete their editing. The second and more feasible option requires acquiring the attention of all other users and then convincing them to temporarily stop their work; even in this case one person halts the work flow of everyone else. An additional problem to both options is the fact that incomplete but error free source code could lead to unexpected behavior, especially so if the feature that is being tested interacts with an incomplete feature. This could results in erroneous testing results or confusion regarding the intended functionality of the program. In either case this might eventually lead to a stifling working environment where testing changes is frowned upon because of the disruptions to work flow. Increasing the number of team members will lead to an increase in disruptions. Automatic code integration therefore leads to a work flow that is jerky; work must be ceased and resumed by everyone to allow one person to test their changes.

The second code integration method is manual integration where any team member can decide to commit a file or files to a master copy from which the program is compiled. Making a commit has the same problem as automatic code integration in that the source code would have to be brought to an error free state however there are a few benefits compared to automatic code integration; compiling the program could be done any number of times by anyone without further disruption to work flow, and since only a number

of files need to be committed the disruption might only affect a fraction of the team instead of everyone. The downside of a master copy is the need to browse two sets of files, view and master copies; in order to verify whether the compilable source code is different from the code in the editor view, this scenario might arise if testing produces unexpected results due to interaction with another part of the program that is different from what is visible in the editor. Consider the following example: Alice and Bob are both on the same project but different files. Alice makes a commit, compiles and runs the program. While testing a feature she implemented, Alice runs into unexpected behavior and checks the source code in the master copy. She finds that a feature in a different file has an old implementation that was changed in the specification. Querying Bob she finds out that the new implementation is only now being worked on and has not been committed yet. This example might be somewhat far-fetched but illustrates a disconnect in awareness that could result if the editor view were to be decoupled from compilable code in this manner.

Error-mediated code integration considers error-generating code written by other users to be outside the IDE feedback functions. No syntax error marks will appear nor will code completion trigger for these sections. Once a part written by some other user is error-free, it is integrated automatically into the local working copy with full IDE features. The main benefit here is the automation; focus need not be shifted to manually integrating changes by other users and thus work can continue uninterrupted.

The previous examples all relied on the IDE not compiling source code that contains syntax errors. To ease compilation in a collaborative environment it is possible to allow compilation with errors but depending on the IDE this could lead to a crash on an error-containing line.

5.3 Awareness of source code changes

A problem of traditional is software development is awareness of changes in source code. Changes made by one user are not accessible by others until the changes have been committed to a repository and awareness of changes is not achieved until the other users access the repository to either pull files or commit their own changes. These changes in source code can interact with unpredictable results or outright conflict each other. The question then is how to maintain awareness of changes made by other users that might impact implementation and testing of new code?

Traditional software engineering has a number of tools, such as FASTDash (Biehl 2007), Syde (Hattori 2010) and CollabVS (Hedge 2008) that attempt to address this problem by highlighting conflicts in real-time thus bypassing the repository interaction threshold for information transfer. In real-time collaborative coding however source code conflicts should not occur since concurrently editing the same lines without being aware of it is

impossible as long as synchronization is maintained. The real-time nature of editing however allows for changes in source code to occur constantly and invisibly that would not occur when synchronizing with a repository. For example let us assume the following scenario: Alice, Bob and Charlie are collaboratively editing source code in real-time. Alice and Bob edit the same file and thus are aware of each other's work. Charlie however is editing a different file but one that implements a feature that Alice utilizes in the source code she is editing. Alice is not maintaining awareness of what Charlie is doing nor does the IDE inform her that Charlie is editing a function she calls. Alice compiles and proceeds to test her changes but runs into unexpected behavior. A problem solving session with Bob proves unfruitful until Charlie is asked about the issue and it is revealed that Charlie's edits have had an undesirable impact on the functionality of the program. In traditional software engineering Alice could have been aware of the changes made by Charlie when she synchronized with a repository but with real-time collaborative coding Charlie's work has "slipped under the radar" of Alice's awareness.

Crude methods of informing others include verbal and text notifications such as tagging a part of source code. Verbally informing someone is a singular event that can be forgotten even if the recipient acknowledges it. Tagging source code requires that others discover and read the note. Such methods have multiple points of failure such as human memory, discovering a notification and relevance to others (there might not be any). Therefore a superior solution would be one that could evaluate the relevance of changes made in one part of the system to changes made in another part, and would automatically provide a warning. Such as a system would best be implemented as an automated feature of an IDE that would monitor changes happening across the project and provide an alert to users who are editing source code that calls functions being edited by another user. This would alert users to the need to communicate on the changes being made and the impact they might have on others.

5.4 Studies in effects of collaboration on coding

A number of controlled experiments on pair programming have been conducted by Nosek (1998), Williams et al. (2013) and Nawrocki et al. (2001). Unfortunately their findings and testing methods are conflicting at points, additionally in Williams' experiment subjects were not directly monitored and as such it is not possible to tell whether they had external assistance. Where Nosek and Williams found that pair programming sped up development, Nawrocki reported that there was no significant change in development speed between pair and solo programmers on most assignments. Nosek and Nawrocki also had differing definitions on when an assignment was complete; Nawrocki's experiments were regarded complete when a set of automated tests were fully passed whereas Nosek's were apparently handed in with different levels of functionality.

Lui et al. (2006) studied these discrepancies further in their study that focused on pairs that consisted of either novices or experts versus solo programmers performing repeat-

programming. Unlike Nosek and Nawrocki's experiments where pairs were formed randomly, and Williams' where groups were formed with an even skill level in mind, Lui deliberately split the teams into pairs of novices and pairs of experts. Shown in figure 10 below is the experiment pitting pairs and solos against each in repeating tests.

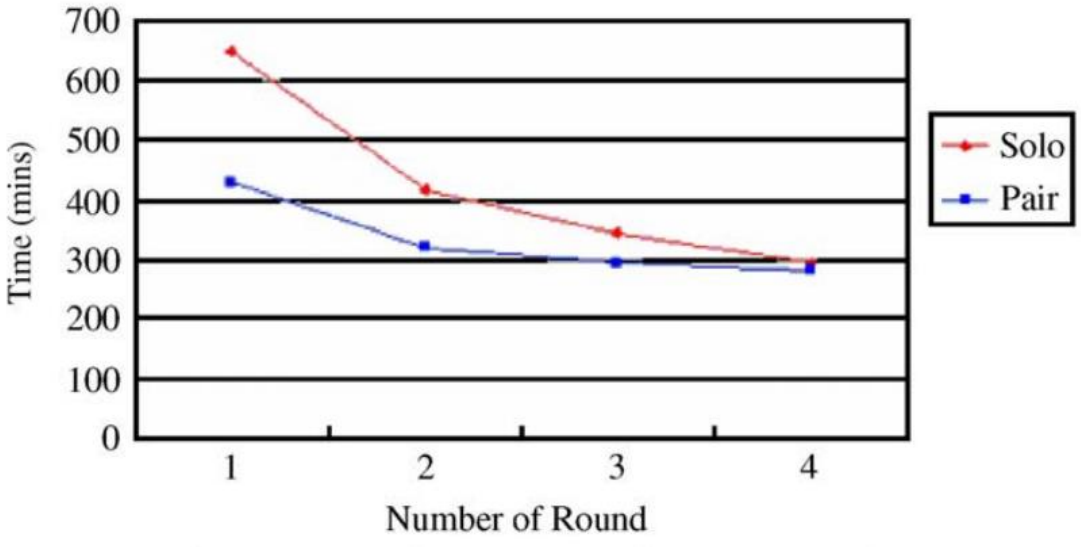

Figure 10 *Repeat-programming (8 groups of 3 "similar-capacity" members, 1 pair and 1 single) (Lui et al. 2006).*

Repeat-programming involved having the pairs and solo programmers complete the same assignment multiple times from scratch in an effort to find out how big an effect does pair programming have on completion time. Lui found that while pairs were initially faster than solo programmers, repeating the same problem would eventually narrow the gap between pairs and solos to nothing. They were not able to find a conclusive reason for the discrepancies in the three studies mentioned above, but concluded that repeat-programming illustrated that familiarity with a program eventually lowered the productivity of pairs compared to solo programmers. From their findings they derived two principles for pair programming when the pairs are formed by two novices or two experts. Firstly a pair is faster and produces higher quality software than two individuals when the pair is new to a problem. Secondly the productivity of pair programming can drop when they have previous experience of the same problem and still remember it. These principles lead them to the conclusion that novice pairs against novice solos are more productive than expert pairs against expert solos. Table 3 below presents a pair programming framework derived from these principles by Lui et al. (2006).

Table 3 *Potential pair programming framework for managing inexperienced programmers (Lui et al. 2006).*

Step	Activity
1.	Pair up
2.	Work on design and algorithm and identify patterns of logic
3.	Code and test sub-programs in pair programming
4.	When the pair encounters any sub-program in which same logic has been done in pair before, the pair should split off and two programmers independently code (and test) the sub-program in solo programming
5.	Pair up
6.	Review and perform integration tests
7.	Go back to step (2) until completion of assignments

This framework presents steps that less experienced programmers can take to maximize the benefits of collaboration. More difficult tasks are tackled together and trivial ones can be coded solo with solutions reviewed against each other.

5.5 The impact of cultural differences

Increased global connectivity via the Internet has enabled offshore outsourcing in software development. Wider geographical distances have however brought cultural differences, also known as cultural distance, into a wider consciousness of issues surrounding offshore outsourcing. Two distinct groups that have differing shared values, beliefs, and norms have cultural differences. These shared attributes are also called cultural dimensions. Perhaps the most famous and argued model of cultural dimensions was created by Hofstede and now consists of six dimensions (Hofstede et al. 2010). Table 4 below contains a list and short descriptions of the six dimensions.

Table 4 *Cultural dimensions according to Hofstede et al. (2010).*

Dimension	Description
Power distance (index)	The level of acceptance towards unequal power distribution by the less powerful in organizations and institutions.
Uncertainty avoidance (index)	Society's degree of tolerance for uncertainty and ambiguity. High uncertainty cultures attempt to minimize the possibility of new, surprising, and unknown situations by strict laws and rules.
Individualism / Collectivism	The degree of the integration of individuals into groups. In individualistic societies personal achievements and rights are important. Collectivist societies integrate new members into cohesive in-groups, and offer protection for unquestioning loyalty.
Masculinity / Femininity	The distribution of emotional roles between genders. In masculine societies men behave in a more assertive and competitive way whereas in feminine societies men are more modest and caring. In masculine societies women are also more assertive and competitive but not as much as men.

Long-term orientation / Short-term orientation	Long-term oriented countries favor pragmatic values such as saving, persistence and adaptation to changing circumstances. Short-term oriented countries favor the values of national pride, respect for tradition, and fulfilling social obligations.
Indulgence / Restraint	Society's allowance of a relatively free gratification of basic human desires. Restraint means the suppression and regulation of the gratification of human needs and desires by strict social norms.

The effect of these cultural dimensions should be considered when real-time collaboration is undertaken over a cultural distance. Potential disruptive scenarios arise when certain combinations of collaborative scenarios occur. For example the cultural dimension of power distance when combined with masculinity can manifest together in a negative fashion if a high power distance and masculine culture member were to collaborate with a low power distance and feminine culture member. In such a scenario the more assertive team member might take for granted a leadership role whereas the team member from a more feminine culture could resent an authoritarian approach to collaboration. Many more such combinations exist but unfortunately a closer examination is beyond the scope of this thesis.

6. CORED WEB-IDE

This section contains excerpts from a study made on CoRED, a real-time collaborative web-IDE developed at Tampere University of Technology. The aim of the original research was to study the effects of learning in a collaborative coding environment, and how real-time collaborative coding can be visualized.

CoRED and its successor MIDeaaS are web-IDEs created for making web applications. MideaaS is integrated with a hosting solution for fast deployment of the developed software as a service. MideaaS offers a simple yet powerful web-based tool for implementing and publishing web applications written in Java using Google Web Toolkit (http://www.gwtproject.org/) and an open source web application framework for rich Internet applications called Vaadin (Grönroos 2014). The development of new software only requires a browser and concurrent editing is supported. Hence, several developers can work on the software simultaneously and see all of the edits in real time. They enforce collaboration by having a single repository per project that every user in the project can access. Automatic code integration to a single repository means that merge conflicts do not exist however gains from automatic integration in both developer time and ease of use are seriously impacted by the fact that the project cannot be built if even a single error exists in the source code. In practice this means that if two or more users are coding and even one user wishes to build the project everyone else must either complete or comment out their code so that no errors exist in the source code. This is a disruptive factor on coding since everyone else must divert their work flow to accommodate the person wanting to build the project thus testing becomes a cumbersome and disruptive task.

In order to study learning and knowledge transfer in collaborating teams, two different code camps were organized by the Department of Pervasive Computing at the Tampere University of Technology. These code camps utilized CoRED and MIDeaaS for teaching students web development through team collaboration and coding. In Section 6.1 we present a study of the log data produced during the two camps. Large volumes of data of great variety can be recorded; however, the problem lies in recognizing, displaying, and utilizing the correct data. Different approaches to recording and visualizing data have been taken by others (Wang et al. 2014; Weissgerber et al. 2007; Voinea et al. 2007). Here we focus on the quality and suitability of the data collected for analysis, the use of coding patterns to recognize learning through collaboration, and the detection of programming patterns in the log data.

6.1 CoRED data logging

During the first code camp (Nieminen et al. 2013) data were gathered from the use of the browser based CoRED software (Lautamäki et al. 2012) which enables multiple users to collaborate simultaneously on a web software project. The distinctive features of CoRED are the ability to allow two or more users to edit the same source code file and display those changes in real-time; in addition, edits and other user actions such as creating a new file are logged and can be visualized to enable recognition of patterns in user behavior and ways of working. User actions are explained below. The second code camp (Kilamo et al. 2014) featured a cloud-based IDE called MIDEaaS, which evolved from CoRED. Both of these development environments have been developed at the Tampere University of Technology.

The data gathered in the two code camp events focuses on user actions on files and source code. The data logged during the first code camp are more varied compared to the second, which concentrates almost solely on user edits. Table 5 (below) contains a list of recorded events, event data, and which code camp logged the event. Each log entry is preceded by a 13-digit timestamp providing millisecond precision.

Table 5 *Logged events*

Event	Recorded data	Code camp
New file	File name	1
Delete file	File name	1
Open file	File name, collab id, user id, user name	1
Close file	File name, collab id, user id, user name	1
Add marker	File name, collab id, marker id, marker type, start position, end position	1
Remove marker	File name, collab id, marker id	1
Edit	File name, collab id (code camp 1), user id (code camp 2), user name (code camp 2), insert/delete, position of edit, number of characters inserted/deleted, number of characters in file after edit	1&2
Chat message Created	User id, message	1&2 2
Loaded from disk	File path for project	2

Of the first six events that are exclusive to the first code camp, four contain user information in the form of two id numbers. The collab id is assigned to a text editor window and one user can therefore have multiple collab ids should they have multiple text editors open. A user id is assigned upon login, so if the user logs in again they are assigned a new user id. Both of these types of ids are problematic since their persistence is not guaranteed.

The first four events are not elaborated on, as the information they contain should be self-explanatory. The add and remove marker events record visual pointers, either signifying a code error from the automatic error checker or the cursor or selection of another user. The marker events add a considerable amount of lines to the log files since typing in code manually will produce errors from the Java error checker. The other type of marker, the cursor location of another user, seems redundant considering that edit events log the activity of other users and contain more detailed information. The edit event utilized during the first code camp contains only the collab id as identifying information, which makes distinguishing between users difficult. This deficiency was fixed for the second code camp with the addition of the user name and a persistent id number for each user. The last two events are exclusive to the second code camp and replace the more verbose file events of the first code camp. The created event signifies the start of a project and is the first event in a log file.

Overall, the data recorded into log files is suitable for examining user actions both alone and in collaboration with others. The changes made in recording between code camps reduced the volume and verbosity of data although some of these changes, such as the removal of a new/delete file event from code camp 2 logs, might prove detrimental.

The quality of the log files was analyzed with Excel as the original research publications (Nieminen et al. 2013; Kilamo et al. 2014) made no mention of errors. Analysis of the files found 4 different types of errors, which are listed in table 6 and discussed below.

Table 6 *Types of errors in log files*

Event/Recorded data	**Description of error**	**Code camp**
Entire row, edit events only	Logged data contains duplicate rows.	1 & 2
Timestamp	Within one millisecond a user is logged as having performed multiple edits (inserts and/or deletions).	1 & 2
Number of characters in file	The number of characters in file-property is static for multiple edit events. This occurs independently and also in connection with the above timestamp error.	1 & 2
User id and name	Some edit events do not contain a user id and name.	2

The existence of duplicate lines was known before analysis of the log files began but, since no hard numbers existed, a count was performed. Code camp 1 material contained 18 log files of which 5 had no duplicate lines. The remaining 13 files contained the vast majority (99.8%) of lines logged during code camp 1. The average for duplicate lines per total lines is 0.02 with the minimum value being 0.01 and the maximum 0.07. Code camp 2 produced 7 log files, all of which contained duplicates. The amount of files is smaller because each project's logged edits were combined into a single file per project. Unlike code camp 1, where there were no significant outliers in duplicate lines per total lines, code camp 2 contains two logs that deviate significantly from the rest. Logs 3-7 have an

average of 0.01 duplicate lines per total lines although logs 1 and 2 contain 0.18 and 0.07 duplicate lines per total lines.

An issue with timestamps and edits exists wherein a single user is logged as having performed multiple edits within a millisecond.

The number of characters in file property reflects the total size of the file. It is updated after each edit but an error exists where the count is not properly updated. This causes visualizations that depict changes in file size to appear flat with sudden jumps in size. This could cause a false impression of programming skills, which is explored further in section 6.3 “Different types of visualizations”.

The last issue found in code camp 2 logs is the lack of user id and name for some edit events. The average for missing user ids and names per total lines is 0.03 with a minimum of 0.01 and a maximum of 0.08. No strong correlation existed between project size and this error.

Overall, we conclude that while log recording has problems, the errors present in the existing data are not plentiful enough to threaten the results of the previous studies (Nieminen et al. 2013; Kilamo et al. 2014). We recommend the log recording process undergo a thorough review in order to dispel any uncertainty regarding results.

6.2 CoRED collaboration visualizations

This brief section contains an explanation of the visualizations produced from CoRED and MIDeaaS log data. Figure 11 (below) is made from code camp 1 log data but its form is the same as the ones used to visualize code camp 2 log data.

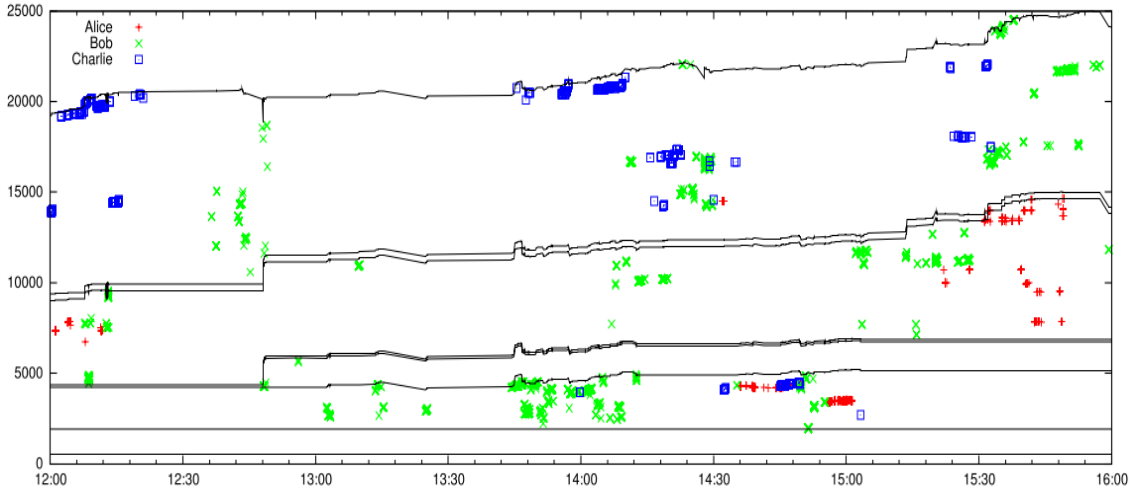

Figure 11 *A four-hour long section from one of the code camp projects (Nieminen et al. 2013).*

The above figure contains a four-hour segment of programming work done on a single project. The axes contain time and number of characters in the project. Grey lines separate different files and show changes in file size as users perform edits. Users' edits are marked with a red plus sign, green x or a blue square as shown in the legend in the upper left hand corner.

The visualizations produced from MIDeaaS log data focus on user actions on source code and interaction with other users. For the sake of readability, visualizations are limited to 2-hour segments as shown in figures 12 and 14. This focus on user-level actions means that observing project-level patterns would require combining several 2-hour segments in order to form an overall picture of project-level progression. Additionally, neither IDE contains version control or issue tracking, thus limiting the options for detecting project level problems.

The currently available data focus on visible actions (edits) performed by users, however, it does not account for other kinds of user co-operation, such as advising done over communication software (Skype etc.) or live discussions. To determine whether this kind of 'invisible' co-operation has occurred, one possible solution could be to record which area of source code is being examined in the IDE. Should two or more users be examining and/or editing the same area of source code, knowledge transfer via problem solving or advising between users becomes possible. Figure 12 (Kilamo et al. 2014) below has been edited to demonstrate this concept.

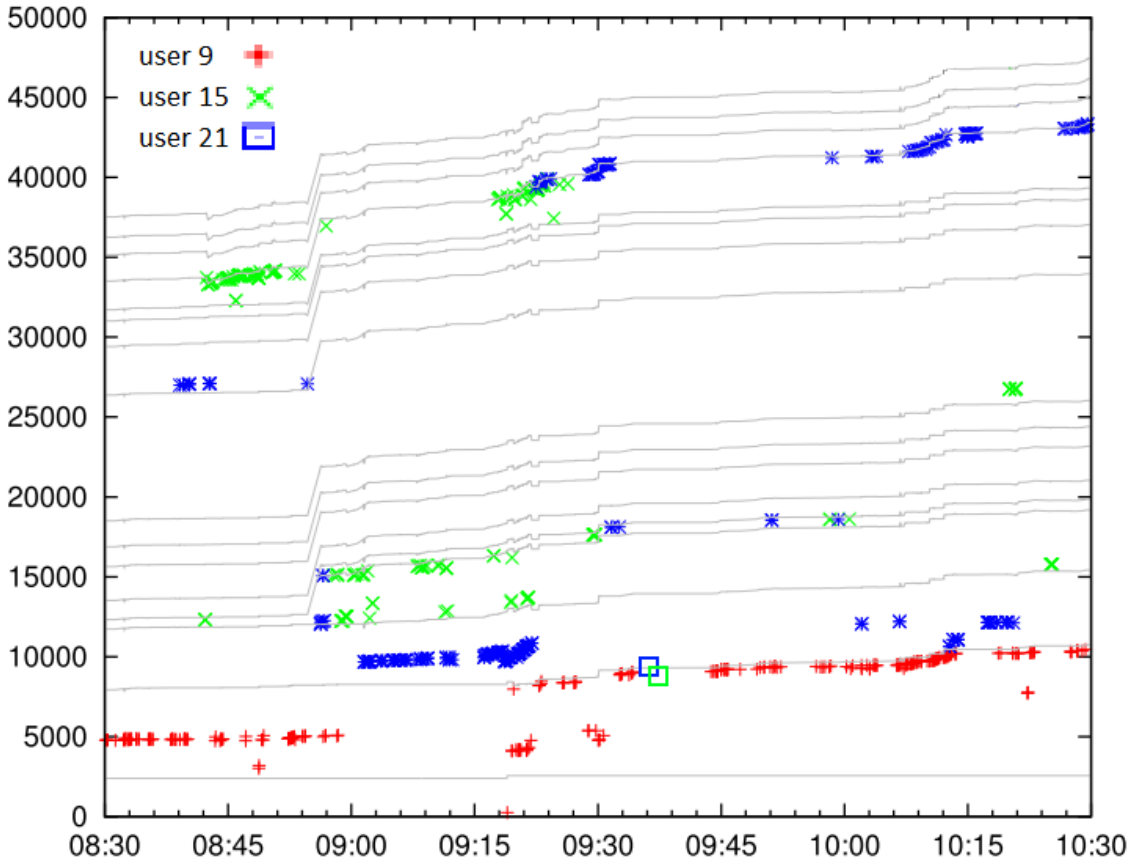

Figure 12 *A typical two-hour-long editing session from one of the teams (Kilamo et al. 2014). Edited to illustrate tracking of each user's focus.*

Here the green and blue squares edited into the picture around the 09:35 mark illustrate the editor views of users 15 and 21 respectively, as they engage in problem solving with user 9. This method of observation is not without its weaknesses, however. Something as simple as a multi-monitor setup where two or more monitors have source code files open (or even one monitor with multiple source code files open) could provide another layer of complexity in determining exactly what file the user is actively following. Another problem with this approach to detecting 'invisible' collaborative activity is the fact that it is difficult to determine whether a user was actually co-operating with others or merely satisfying idle curiosity, unless they perform at least some edits in the observed area of source code. Nonetheless, this approach offers an additional method of detecting collaboration.

6.3 Different types of visualizations

Code size growth data have been examined by others as an indicator of programming skills by comparing the growth data curves between programmers of differing skill levels solving the same programming problems (Wang et al. 2014). In the paper by Wang et al., the two best and worst performing programmers' code growth curves were compared, as shown in Figure 13 (below).

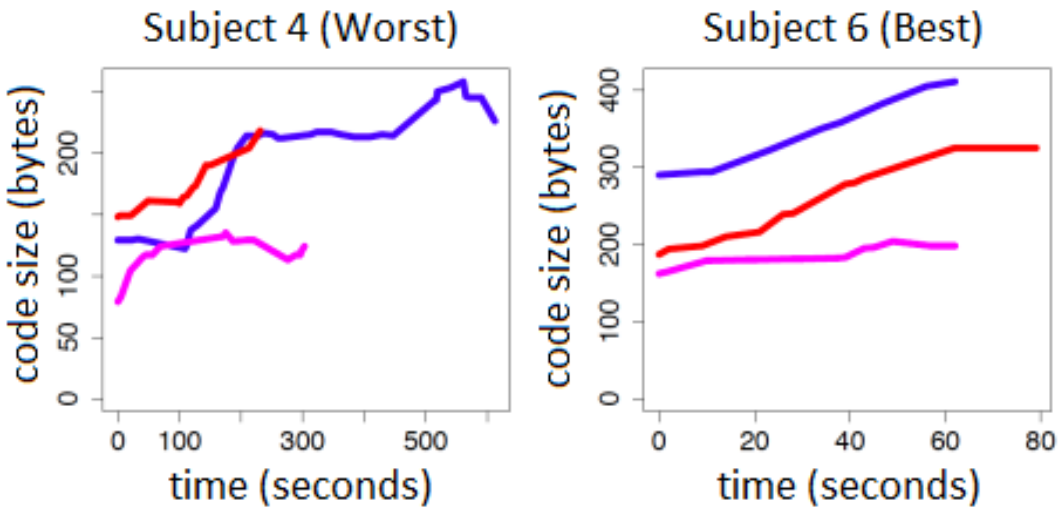

Figure 13 *The traces of the worst and best performers (Wang et al. 2014). Blue graph is problem A, red graph problem B, and orange graph is problem C.*

The figure above contains code size growth curves for two different programmers whose programming skills were evaluated by making them individually solve the same problems. The colors in the figure correspond to different problems. The axes show code size in bytes and time in seconds, although the scales of the plots are different. However the main issue here is the behavior of the code growth rather than the scales of the plots.

The worst performing programmer's curve behaved in an "erratic" way compared to the best programmer, whose code growth curves were "very smooth and generally monotonically increasing". Wang et al. suggest that "Drastic drops imply code deletion indicating that corrections or refactoring occurred. Plateaus suggest "think time" possibly because users were experiencing challenges preventing them from making steady progress".

A direct comparison of code size growth curves between code camp participants cannot be performed since the participants worked on different projects and engaged in collaborative work. However, the approach by Wang et al. could be utilized together with the 'invisible' activity detailed above to observe knowledge transfer and its effectiveness. As an example, let us assume that a user in a collaborative project has been working on a file alone but at some point determines that they are unable to solve a problem on their own. Another team member is asked to provide assistance, which they supply either in the form of advice ('invisible' activity) or coding (edits). The outcome of this interaction could be observed from the code size growth curve of the user who requested help. If the curve behaves in a "smooth and generally monotonically increasing" way after interaction, we can say that collaboration has transferred knowledge successfully. Other possible areas of interest around this kind of scenario would be to examine how successful (does the advisee's code size growth curve indicate hesitation or confidence in their work?) different users are in this kind of situation in different roles (adviser or advisee). To summarize the benefits of the approach described above would help detect collaboration that would

be missed by examining only edits and help determine the successfulness of collaboration.

As an addition to the suggestion of Wang et al. about drastic drops in code size, we propose that a drastic code insertion is either refactoring (if the insertion follows a deletion) or copy and paste programming (if the insertion occurs without a corresponding deletion). Examining edit events by a single user to determine the occurrence of either refactoring or copy and paste programming would not be prudent in this case due to the collaborative nature of the code camp projects. For example, two users working together but examined separately could lead to false conclusions regarding refactoring. Figure 14 (below) contains collaborative work around a drastic change in code size (Kilamo et al. 2014).

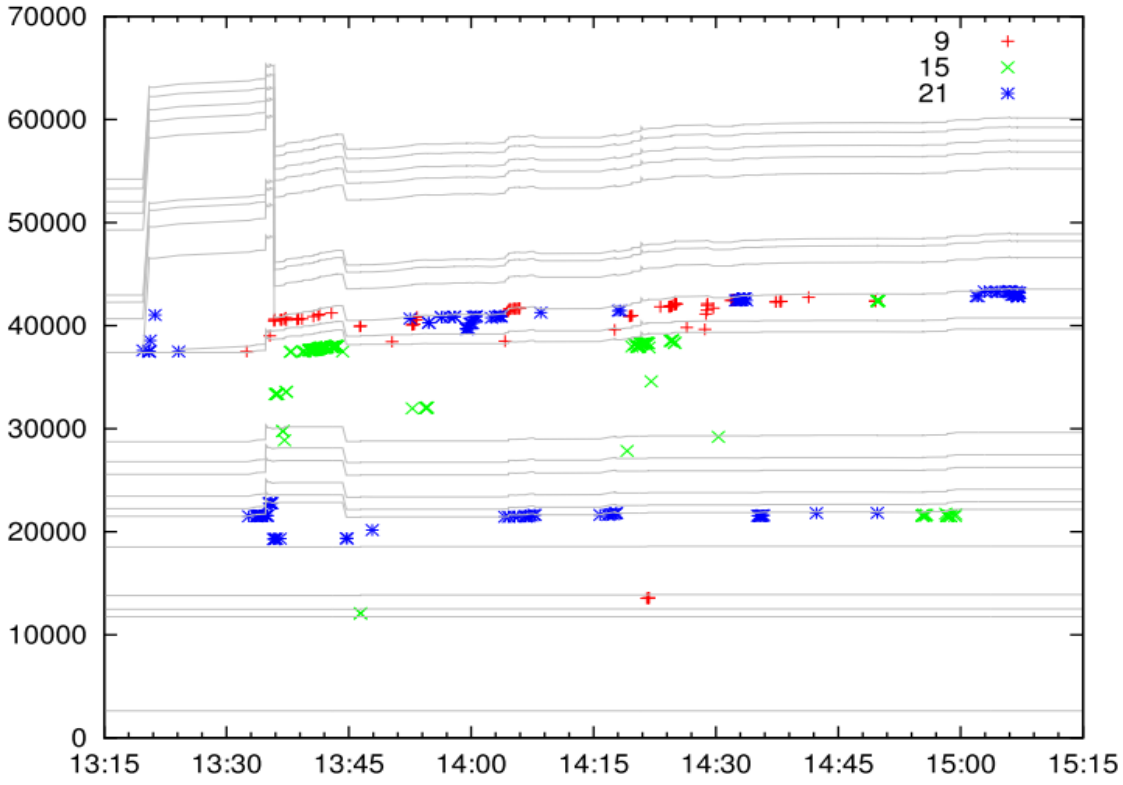

Figure 14 *A typical two-hour-long editing session from one of the teams (Kilamo et al. 2014).*

Here user 21 has inserted a large amount of code around 13:19. However, further work on the file is continued by user 9, whereas user 21 has moved on to work on another file. A task handover has been performed between users 9 and 21, and it is reasonable to assume that some knowledge must have been exchanged via advice regarding the code added by user 21 and possible further work needed on it.

6.4 Detecting a potential anti-pattern

Andrew Koenig first introduced the idea of an anti-pattern (Koenig 1995) in software engineering in 1995. He defines it as follows: "Anti-pattern is just like pattern, except that instead of a solution it gives something that looks superficially like a solution, but isn't one." Whereas a (design) pattern provides a solution to a design problem, an anti-pattern is a solution that appears to be beneficial but contains disadvantages that may negate any potential benefits. Since all existing visualizations produced from MIDeaaS log data focus on measuring and displaying user edits and code size growth, observable anti-patterns are coding, code, or work habit related. The anti-patterns presented below are rudimentary and not collaboration-specific but nonetheless offer an opportunity to detect problematic coding behavior.

Visualization of MIDeaaS log data can be used to detect at least one problematic coding-related anti-pattern: copy and paste programming (Brown et al. 1998), which is the reuse of code, possibly without fully understanding its purpose. The use of this technique is in some cases justified with increased productivity via code reuse. However, should one of the copied parts of code contain an error that is not immediately evident through inspection and/or testing, tracking down and fixing all instances of the copied code can become costly. Copying and pasting can be detected in the visualization graphs as vertical increases in code size that clearly differ in the scale of increase when compared to manually typed code. Fig. 14 above contains an example of a possible copy and paste occurrence, although the earlier edits should also be examined to determine whether refactoring was the case there.

A code-related anti-pattern that is observable from visualizations is the god object (Riel 1996). In programming, a god object breaks good object-oriented design by containing most of the information or methods of the entire program in a single class. In the MIDeaaS visualizations, a god object can be found by comparing file sizes; no exact definition for the size of a god object exists but should one file clearly exceed the size of the others in one project and the class in question be tightly coupled with the rest of the project’s classes, the class should be inspected for the existence of this anti-pattern.

The visualizations produced by MIDeaaS allow not only a view of collaboration within a team but a window to individual coding habits and behavior. This presents a chance to help users by detecting possible bad coding habits.

6.5 Future topics

Invisible activity. The effects of real-time co-operation on learning and knowledge transfer in general are difficult to measure. In order to determine an increase in programming skills in an individual, one would first have to examine behavior over a long time in order

to establish a baseline. Even determining a metric or a set of metrics to evaluate someone's skill is not a trivial task. Assuming that a suite of metrics could be chosen to represent a person's coding performance and a change in behavior (for better or worse) could be detected, the next step would be to determine exactly what caused this change. Pinpointing the exact event or group of events could be impossible; a human being engaging in creative work such as coding is not a glass box state machine with clearly labeled inputs and outputs. Changes in behavior could be a result of a host of factors all taken together. An additional problem is the fact that neither visualizations nor log data reveal the specific type of programming problem that is being solved. Collaboration might transfer knowledge that helps in solving a specific problem but another type of problem could again hinder a programmer. It is for the above reasons that we suggest that a combination of user edit information and 'invisible' activity should be used to study the effectiveness of collaboration. In collaboration situations, attention should be paid to the following facts: the time spent collaborating, its effectiveness (was work continued by the original programmer?) and any recurring pattern in collaborators (do some people mesh together better than others?).

Changes in recorded data between code camps. The original research group chose to discard six logged events: new, delete, open and close file, in addition to add and remove marker. These events were possibly deemed not informative enough to provide insight into knowledge transfer among users. It is also possible that the file-related events were merely added to help in visualizing log files. Of the six discarded events, new and delete file could help provide information on a project level about library additions or refactoring. This could help clearly differentiate between adding a library file and pasting a large amount of code.

Collaboration thresholds. A possibly interesting avenue of further study would be to examine user actions after launching the application and especially situations where the application terminated with an error. A per user threshold for collaboration could be explored by examining possible patterns in a user seeking collaboration to help solve a problem with an application experiencing errors. This approach should be combined with that described in the first sub-section of this section to maximize the amount of data available. MIDeaaS does not currently support deploying the application if another user is in the middle of editing code. This problem would have to be solved first as currently deploying and running the project is on the whole a group effort.

6.6 Results of collaboration

The original research group judged the applications produced by the project groups based on the idea, the overall complexity, and quality of implementation. Figure 15 below shows the results of the second code camp (Kilamo et al. 2014).

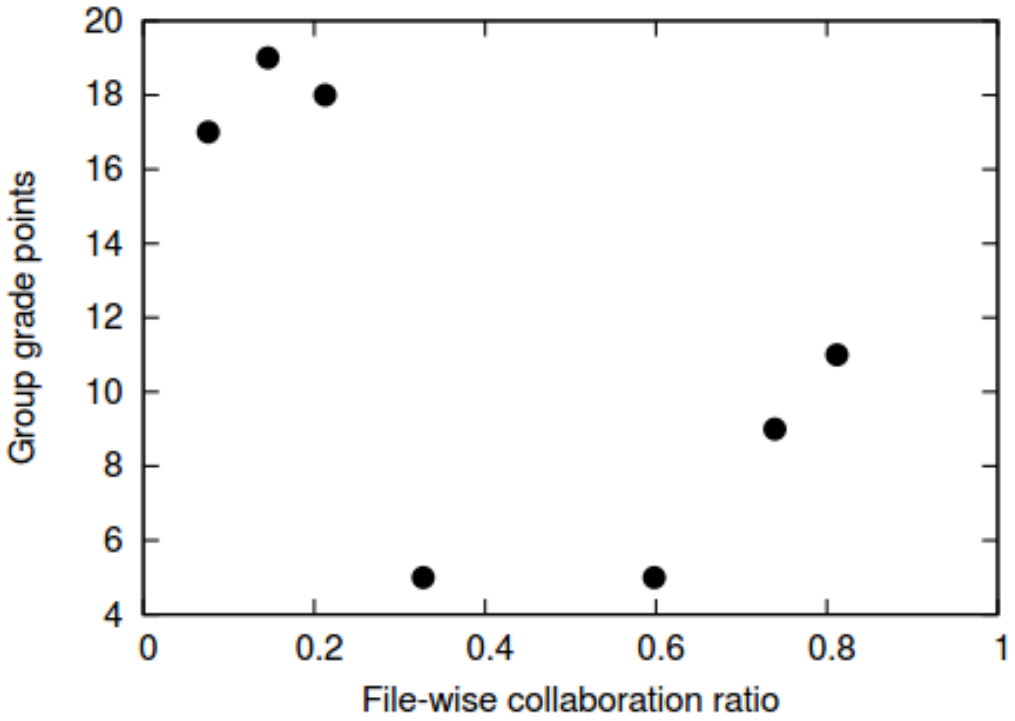

Figure 15 *Group grade points in relation to the file-wise collaboration ratio for each of the groups in the code camp (Kilamo 2014).*

Each group was given a score by the code camp facilitators; the best project received seven points and the worst only one point. The figure above links these points to the file-wise collaboration ratio which measures the amount of collaborative edits done in the same code file within 30 seconds out of overall project collaborative edits. The groups with the highest score also collaborated least, possibly indicating that real-time collaborative coding is not a method with a positive effect. These results are however muddled by the fact the teams were shuffled twice during the code camp; each shuffle had one person from the team remain to instruct the newcomers on the project though this person was not the same one for each shuffle. It is possible that the shuffles had a negative effect on collaboration itself as new team members had to spend time becoming familiar with others within the team and commit to a project they might not be interested in.

6.7 User impressions of collaborative coding

A code camp event was organized for CoRED and its successor MIDeaaS respectively. Surveys were conducted before and after each event to find out the participants' impressions and experience on using a web-IDE, collaboration, and expectations in general. The following collaboration related questions were put to the users after the code camps:

- How did you find the group programming aspects?
- What was your impression on distributed group programming via shared IDE? Did it support your collaboration?

- Did you like the group shuffle experiments (the one in which you worked for or guided an another group)

The participants of both code camps had mostly positive experiences regarding collaboration and group programming. Since the programming skills within the groups were varied some felt that they had learned new skills, improved skills in dividing responsibilities and tasks were also noted. Continuous communication was also described as an important element of successful collaboration. The most common negative answer related to the IDE's enforcement of no compiling with errors which made testing changes more difficult if concurrent editing was happening.

7. DISCUSSION AND CONCLUSIONS

The technical problems and solutions considered in Section 4 highlight the additional difficulties in implementing real-time collaborative coding in an IDE. Between document synchronization and code integration, synchronization is the more trivial issue with no clear "winner". Operational Transformation is a known solution to the synchronization problem. Fraser (2009) provides an alternative implementation in the form of differential synchronization however an empirical test between OT and differential synchronization is outside the scope of this work.

The more interesting technical issue of code integration has had two implementations so far: automatic and error-mediated. Automatic integration is the simpler implementation which however has the negative effect of constantly disrupting group work when compilation is desired. A way around this issue could be to allow compiling with errors however such an approach is more on lines of curing the symptoms rather than the cause of the illness. Compiling with errors introduces its own problems to testing changes as the instability of the compiled program could lead to faulty testing results. In comparison error-mediated integration (Goldman 2012) appears to be a more elegant solution. It allows for individual compilation without disrupting group work. It too suffers from a problem which is admittedly lesser in impact: the constant background integration of source code means that awareness of changes made in other parts of the source code, that might affect the part being tested, is limited. This could also cause some surprising testing results. Nevertheless error-mediated integration comes out on top when compared against automatic integration as the more work flow friendly solution. Further work on this method of integration is required to maintain user awareness on changes being made in other parts of source code in order to avoid anomalous testing results.

Real-time collaborative coding has the possibility to be highly effective in pair programming, and classroom environments. The results found by Lui et al. (2006) highlight the importance of collaboration between novice programmers in tackling difficult problems. This means that real-time collaboration has uses in specifically targeting challenging programming tasks and in increasing the knowledge spaces of programmers. Experts however are less likely to benefit from collaboration as they are likelier to have a pre-existing solution that can be utilized without consulting another programmer.

Real-time collaboration in programming in general requires that IDEs provide users with greater awareness on what each user is doing. Often changes made elsewhere can have an impact in a different part of the program which due to the distributed and concurrent nature of working can be missed if information is not being passed properly. It is not clear whether simple text based notifications left by users themselves are enough to maintain awareness of other users' actions. The flow of information in an office environment can

be subtler: an overheard conversation can trigger a response in a way that notifications left into source code might not. A more effective system might be one that monitors changes being made to source code and automatically notifies users if their current target of work is being called elsewhere in source code and utilized by code being edited by some other user. This could help initiate a flow of information between users and thus avoid unexpected behavior during testing. It is unfeasible to expect users to manually constantly check whether the code they're editing is being affected by changes being made elsewhere. Changes made to affecting code non-concurrently highlight the need for a history feature in order to spot possible non-desirable effects.

The CoRED study provides a view into visualizing collaboration in a concrete form. Further refinements to the visualizations could help show how programmers of different skill levels cope with problem situations. Real-time collaborative IDEs in general require more work in the areas of awareness and smooth code integration in order to enable seamless collaboration that is not negatively impacted by technical solutions or lack thereof.

8. SUMMARY

In this thesis we studied both technical challenges and solutions to implementing real-time collaborative coding in an IDE, and the suitability of adopting real-time collaborative coding in different models of programming. In the topic of document synchronization there is no clear empirically demonstrated advantage to either Operational Transformation or differential synchronization. Both have been successfully implemented in real-time collaborative editors and IDEs. On source code integration of the two currently existing methods, automatic and error-mediated, it is the latter which has the least impact on group work though it is technically a more demanding solution. Automatic code integration suffers from disrupting group work while the downside to error-mediated integration is uncertainty in awareness of changes in compiled source code.

From a group work suitability point of view real-time collaborative coding was found to fit well into pair programming and classroom environments where the participants are both either novices on the subject matter or one is a novice and the other an expert. When both participants are experts, collaboration is unlikely to provide either a gain in development speed or an increase in shared knowledge. Fitting real-time collaborative coding into outsourcing is a more challenging task as with greater differences in time zones comes greater differences in cultures which requires careful preparation to avoid misunderstandings due to cultural differences. Finally the importance of awareness of other users' actions was considered. The concurrent nature of real-time collaborative editing means that IDEs must support information flow at a high level between users so that awareness of changes being made in the system can be maintained by all participants. Current IDEs provide only rudimentary note and chat tools. We recommend an automated system which provides alerts when changes are being made to source code that is called elsewhere and is undergoing modification by another user. Non-concurrent changes also highlight the need for a history feature that enables retrospection.

REFERENCES

K. Beck, C. Andres. (2004). Extreme Programming Explained: Embrace Change, 2nd ed. Addison-Wesley Professional.

J. T. Biehl, M. Czerwinski, G. Smith, G. G. Robertson. (2007). FASTDash: a visual dashboard for fostering awareness in software teams. Proceedings of CHI 2007 (25th SIGCHI Conference on Human Factors in Computing Systems), ACM, pp. 1313–1322.

B. Boehm. (2006). A view of 20th and 21st century software engineering. ICSE ‘06 Proceedings of the 28th international conference on Software engineering. New York, USA: AC, pp. 12-29.

W. Brown, R. Malveau, S. McCormick, T. Mowbray. (1998). AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, ltd.

A. Cockburn, L. Williams. (2000). The Costs and Benefits of Pair Programming. Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2000).

P. Dillenbourg. (1999). What do you mean by collaborative learning? Collaborative-learning: Cognitive and Computational Approaches. Oxford: Elsevier, pp.1-19.

C. A. Ellis, S. J. Gibbs. (1989). Concurrency Control in Groupware Systems. SIGMOD '89 Proceedings of the 1989 ACM SIGMOD international conference on Management of data, pp. 399-407.

D. C. Engelbart, W. K. English. (1968). A research center for augmenting human intellect. In Proc. Fall Joint Computer Conference, Part I - AFIPS, page 395. ACM Press.

N. V. Flor. (1998). Side-by-side collaboration: a case study. International Journal of Human-Computer Studies, Volume 49, Issue 3, pp. 201–222.

N. V. Flor. (2006). Globally distributed software development and pair programming. Communications of the ACM, Volume 49 Issue 10, pp. 57-58.

N. Fraser. (2009). Differential synchronization. DocEng '09 Proceedings of the 9th ACM symposium on Document engineering, pp. 13-20.

M. Goldman. (2012). Software Development with Real-Time Collaborative Editing.

M. Goldman, G. Little, R. C. Miller. (2011). Collabode: collaborative coding in the browser. CHASE '11 Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering, pp. 65-68.

M. Grönroos. (2014). Book of Vaadin. Uniprint.

L. Hattori, M. Lanza. (2010). Syde: A Tool for Collaborative Software Development. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Volume 2, pp. 235 -238.

R. Hegde, P. Dewan. (2008). Connecting programming environments to support ad-hoc collaboration. Proceedings of ASE 2008 (23rd IEEE/ACM International Conference on Automated Software Engineering), IEEE CS Press, pp. 178–187.

G. Hofstede, G. J. Hofstede, M. Minkov. (2010). Cultures and Organizations: Software of the Mind, 3rd ed. New York: McGraw-Hill.

T. Kilamo et al. (2014). Knowledge transfer in collaborative teams: experiences from a two-week code camp. Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014), ACM, New York, USA, pp. 264-271.

A. Koenig. (1995). Patterns and antipatterns. Journal of Object-Oriented Programming 8 (1), pp. 46-48.

J. Lautamäki et al. (2012). CoRED - browser-based collaborative real-time editor for Java web applications. Proceedings of The ACM Conference on Computer Supported Cooperative Work, Seattle.

K. M. Lui, K. C. C. Chan. (2006). Pair programming productivity: Novice–novice vs. expert–expert. International Journal of Human-Computer Studies, Volume 64, Issue 9, pp. 915–925.

K. M. Lui, K. C.C. Chan, and J. T. Nosek. (2008). The Effect of Pairs in Program Design Tasks. IEEE Transactions on Software Engineering, vol. 34, no. 2, March/April.

J. Nawrocki, A. Wojciechowski. (2001). Experimental evaluation of pair programming. Proceedings of the 12th European Software Control and Metrics Conference, London, April, pp. 269–276.

A. Nieminen et al. (2013). Collaborative coding environment on the web: a user study. Developing cloud software: algorithms, applications, and tools, TUCS General Publication Nr. 60, pp. 275-300.

J. T. Nosek. (1998). The Case for Collaborative Programming. Communications of the ACM, vol. 41 no. 3, pp. 105-108.

M. Rantala, J. Soini, T. Kilamo. (2015). Gathering Useful Programming Data: Analysis and Insights from Real-Time Collaborative Editing. MIPRO 2015, May 25-29, Opatija, Croatia.

A. J. Riel. (1996). Object-Oriented Design Heuristics. Boston: Addison-Wesley.

J. Roschelle, S. Teasley. (1995). The construction of shared knowledge in collaborative problem solving. Computer supported collaborative learning, Berlin, Germany: Springer Verlag, pp. 69-197.

G. Stahl, T. Koschmann, & D. Suthers. (2006). Computer-supported collaborative learning: An historical perspective. Cambridge, UK: Cambridge University Press. In R. K. Sawyer (Ed.), Cambridge handbook of the learning sciences (pp. 409-426).

C. Sun, Y. Yang, Y. Zhang, D. Chen. (1996). Distributed concurrency control in real-time cooperative editing systems. Concurrency and Parallelism, Programming, Networking, and Security Lecture Notes in Computer Science Volume 1179, pp. pp 84-95.

L. Voinea, A. Telea. (2007). Visual data mining and analysis of software repositories. Computers and graphics, vol. 31, pp. 410-428.

Y. Wang, P. Wagstrom, E. Duesterwald, D. Redmiles. (2014). New opportunities for extracting insights from cloud based IDEs. ICSE Companion 2014 Companion Proceedings of the 36th International Conference on Software Engineering, pp. 408-411.

P. Weissgerber, M. Pohl, M. Burch. (2007). Visual data mining in software archives to detect how developers work together. Fourth International Workshop on Mining Software Repositories.

L. Williams, R. R. Kessler, W. Cunningham, R. Jeffries. (2000). Strengthening the Case for Pair Programming. IEEE Software. IEEE Software, July–Aug. 2000.